Mock exam 1

ETH Zurich

Date: 9./10. November 2009

Name: ______________________________

Group: ______________________________

1 Terminology (8 points)

Put checkmarks in the checkboxes corresponding to the correct answers. Multiple correct answers are possible; there is at least one correct answer per question. A correctly set checkmark is worth 1 point, an incorrectly set checkmark is worth -1 point. If the sum of your points is negative, you will receive 0 points.

Example:
Which of the following statements are true?

1.
   a. Classes exist only in the software text; objects exist only during the execution of the software. ☒
   b. Each object is an instance of its generic class. ☐
   c. An object is deferred if it has at least one deferred feature. ☐

Solution

Which of the following statements are true?

1. A command...
   a. is a query that is not implemented as an attribute. ☐
   b. may modify an object. ☒
   c. may appear in the precondition and the postcondition of another command but not in the precondition or the postcondition of a query. ☐
   d. may appear in the class invariant. ☐

2. A query...
   a. may be used as a creation procedure. ☐
   b. may be implemented as a routine. ☒
   c. may appear in the precondition and the postcondition of another query but not in the precondition or the postcondition of a command. ☐
   d. may appear in the class invariant. ☒

3. A class...
   a. is the description of a set of possible run-time objects to which the same features are applicable. ☒
   b. can only exist at runtime. ☐
   c. cannot be declared as expanded; only objects can be expanded. ☐
   d. may have more than one creation procedure. ☒

4. Immediately before a successful execution of creation instruction with target `x` of type `C`...
   a. `x = Void` must hold. ☐
   b. `x /= Void` must hold. ☐
   c. postcondition of creation procedure may not hold. ☒
   d. precondition of creation procedure may not hold. ☐

5. Immediately after a successful execution of creation instruction with target `x` of type `C`...
   a. `x = Void` must hold. ☐
   b. postcondition of creation procedure may not hold. ☐
   c. precondition of creation procedure may not hold. ☒
   d. object attached to `x` satisfies the invariant of `C`. ☒

2 Digital root (10 points)

The *digital root* (Quersumme) of a number is found by adding together the digits that make up the number. If the resulting number has more than one digit, the process is repeated until a single digit remains.

Example input and output

Input	Digital root
123	6
5720	5
99999999	9
8	8

Your task in this problem is to implement a function that, given a non-negative number, calculates the digital root and returns it as the result. Fill in the body of function *digital_root* below. Your implementation should work with *INTEGER* objects only. You might find the following two operators of class *INTEGER* useful: \\ (modulo) and // (integer division).

Solution

```
digital_root (a_number: INTEGER): INTEGER
    -- Digital root (Quersumme) of 'a_number'
  require
```

```
    a_number_within_range: a_number >= 0 and a_number <= a_number.max_value
  local
    number: INTEGER
  do
    from
      Result := a_number
    invariant
      result_non_negative : Result >= 0
    until
      Result < 10
    loop
      from
        number := Result
        Result := 0
      invariant
        -- 'Result' is a sum of i lower digits of 'old Result'
        -- 'number' contains n - i upper digits of 'old Result'
      until
        number = 0
      loop
        Result := Result + (number \\ 10)
        number := number // 10
      variant
        number
      end
    variant
      Result
    end
  end
```

3 Design by Contract (10 Points)

Class *PERSON* is part of a software system that models marriage relations between persons. The following rules do not necessarily have universal value but describe a particular set of rules for marriage at a particular time and place in the past, e.g. Canton Zürich 1900:

1. A person cannot be married to himself/herself.
2. If a person X is married to a person Y, then Y is married to X.
3. In order for a person X to be able to marry a person Y, neither X nor Y may be already married.

Your task is to fill in the contracts of the class (preconditions, postconditions and class invariant) according to the specification given. You are not allowed to change the class interfaces or any of the already given implementations. Note that the number of dotted lines does not indicate the number of necessary code lines that you have to provide.

Solution

```

class
    PERSON

create
    make

feature -- Access

    name: STRING
            --Person's name

    spouse: PERSON
        -- Spouse if a spouse exists, Void otherwise

feature -- Creation

    make (n: STRING)
            -- Create a person with a name
        require
            n_exists_and_not_empty: n /= Void and then not n.is_empty
        do
         -- Create a copy of the argument and assign it to name
            name := n.twin
        ensure
            name_set: n.is_equal (name)
            not_married_yet: not is_married
        end

feature -- Status report

    is_married: BOOLEAN
            -- Is current person married?
```

```
        do
            Result := (spouse /= Void)
        ensure
            is_married: Result = (spouse /= Void)
        end

feature {PERSON} -- Implementation

    set_spouse (p: PERSON)
            -- Set spouse to p
        require
            p_exists: p /= Void
            p_not_current: p /= Current
            current_not_married: not is_married
            target_maybe_married: p.spouse = Void or p.spouse = Current
        do
            spouse := p
        ensure
            spouse_set: spouse = p
            is_married: is_married
        end

feature -- Basic operations

    marry (p: PERSON)
            -- Get married to p
        require
            p_exists: p /= Void
            p_not_current: p /= Current
            current_not_married: not is_married
            target_not_married: not p.is_married
        do
            set_spouse (p)
            p.set_spouse (Current)
        ensure
            current_is_married: is_married
            other_is_married: p.is_married
            current_spouse_is_p: spouse = p
            p_spouse_is_current: p.spouse = Current
        end

invariant
        name_exists_and_not_empty: name /= Void and then not name.is_empty
        marriage_semantics: is_married = (spouse /= Void)
        marriage_not_reflexive: spouse /= Current
        marriage_symmetric: is_married implies (spouse.spouse = Current)

end
```

4 Doubly linked lists (14 points)

In the lecture you have been taught about singly linked lists, which allow to move through the list in one direction. In this task you have to implement a data structure called a *doubly linked list*, which should allow moving in both directions through the list. The structure consists of two classes: INTEGER_LIST_CELL and INTEGER_LIST. An object of type INTEGER_LIST_CELL holds an INTEGER as the cell content and has a previous and a next reference to two other objects of type INTEGER_LIST_CELL. By attaching the previous and next references correctly, two or more cells can be connected to form a list. The class INTEGER_LIST offers functionality to access the first and the last cell of a list, to add a new cell at the end, and to look for a specific value in the list. In Figure 1 you see a drawing of a doubly linked list.

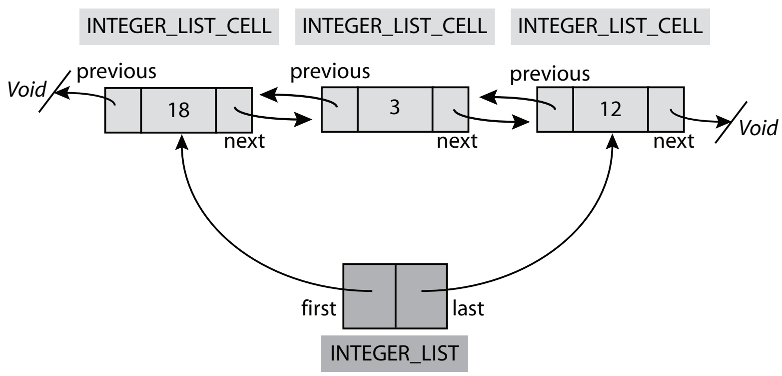

Figure 1: Doubly linked list

Read through the class INTEGER_LIST_CELL in Listing 2. You will need the features of this class for the rest of the task.

1. Implement the feature *extend* of class INTEGER_LIST (see Listing 1). This feature takes an INTEGER as argument, generates a new object of type INTEGER_LIST_CELL with the given INTEGER as content and puts the new cell at the end of the list. Make sure that your implementation satisfies the given postcondition of the feature.

2. Implement the feature *has* of class INTEGER_LIST (see Listing 1). This feature checks if the value it receives as argument is contained in any cell of the list. In the example of Figure 1, the first cell contains the value 18, the second cell contains the value 3, and the third one contains the value 12.

Solution

Listing 1: Solution class *INTEGER_LIST*

```
class
  INTEGER_LIST

create
  make_empty
```

```
feature -- Initialization

  make_empty is
      -- Initialize the list to be empty.
    do
      first := void
      last := void
      count := 0
    end

feature -- Access

  first : INTEGER_LIST_CELL
      -- Head element of the list, Void if the list is empty

  last: INTEGER_LIST_CELL
      -- Tail element of the list , Void if the list is empty

feature -- Element change

  extend (a_value: INTEGER) is
      -- Append a integer list cell with content 'a_value' at the end of the list .
    local
      el: INTEGER_LIST_CELL
    do
      create el.set_value (a_value)
      if empty then
        first := el
      else
        last . set_next (el)
        el . set_previous (last)
      end
      last := el
      count := count + 1
    ensure
      one_more: count = old count + 1
      first_set : count = 1 implies first.value = a_value
      last_set : last . value = a_value
    end

feature -- Measurement

  count: INTEGER
      -- Number of cells in the list

feature -- Status report

  has (a_value: INTEGER): BOOLEAN is
      -- Does the list contain a cell with value 'a_value'?
    local
      cursor: INTEGER_LIST_CELL
    do
```

```
      from
        cursor := first
      until
        cursor = Void or Result
      loop
        if cursor.value = a_value then
          Result := True
        end
        cursor := cursor.next
      end
    end

  empty: BOOLEAN is
      -- Is the list empty?
    do
      Result := (count = 0)
    end

end
```

Listing 2: Class *INTEGER_LIST_CELL*

```
class INTEGER_LIST_CELL

create
  set_value

feature -- Access

  value: INTEGER
      -- Content that is stored in the list cell

  next: INTEGER_LIST_CELL
      -- Reference to the next integer list cell of a list

  previous: INTEGER_LIST_CELL
      -- Reference to the previous integer list cell of a list

feature -- Element change

  set_value (x: INTEGER) is
      -- Set 'value' to 'x'.
    do
      value := x
    ensure
      value_set: value = x
    end

  set_next (el: INTEGER_LIST_CELL) is
      -- Set 'next' to 'el'.
    do
      next := el
    ensure
```

```
      next_set: next = el
    end

  set_previous (el: INTEGER_LIST_CELL) is
      -- Set 'previous' to 'el'.
    do
      previous := el
    ensure
      previous_set: previous = el
    end

end
```